Factoring Benchmark for SAT-solvers

Tuomo Pyhälä

June 24, 2004

Abstract

In this paper we develop a SAT-solver benchmark based on the factoring problem from number theory. We develop a generator of hard factoring instances, which can be arbitrarily large and still contain structure. The instances are experimentally indeed found to be difficult to solve as we study the performance of different SAT-solvers.

1 Introduction

Propositional satisfiability solvers have been lately adapted to different tasks [14, 3, 5, 4, 2] and remarkably improved. The Boolean circuit satisfiability solvers [13] are rare. However, they have the advantage of having more complete knowledge of the problem structure, which the SAT solvers do not have.

In this paper we create a program generating hard problem instances and compare different SAT-solvers capability to solve them. The hard instances are based on the factoring problem, which was proposed by Cook and Mitchell in [7]. Our approach to generation of the instances is to take a industrial design multiplier circuit, and sets the given number as the output. Then we employ SAT-solver to deduce backwards the inputs, which are the factors we were looking for. See Fig. 1.

Hard SAT instances are useful in benchmarking and improving the performance of the different SAT solvers. Traditionally there have been instances with structure based on for example verification of circuits. However, these instances have been static in size. On the other hand, there has been randomized

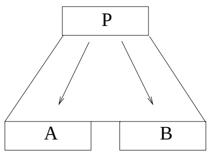

Figure 1: Our approach, we solve the A and B satisfying P = AB backwards, the arrow showing the direction of the deduction

instances, which do not have any structure but can be scaled arbitrarily. Our benchmark tries to combine the properties in a new way. The problem has the structure from the multiplier circuits and on the other hand we may scale the circuits to arbitrary size and find numbers to be factored of arbitrary size.

We believe that factoring is a hard problem to solve. Hardness of factoring is essential to security of the RSA algorithm, because breaking RSA is at most as hard as solving factoring (loosely defined as inverse of multiplication) [19]. RSA is widely used cryptographic cipher and is used for example in secure e-mail, secure online banking etc. Therefore we believe that it is not easy to solve the problem.

Currently algorithms based on the number theory are able to factor numbers few hundred bits wide. The properties of the number to be factored severely affect the time required. The product of two primes of about equal size should be a hard instance [19]. We believe that SAT solver based approach is not as effective as these problem specific algorithms. However, we use the kind of numbers, which are also hard to solve for these algorithms.

2 Theoretical Background

Factoring is a problem where one tries to find the primitive factors of a number. For example, primitive factors of fifty are two, five and five, because $50 = 5*5*2$. Factoring is assumed to be hard, because no polynomial time algorithm has been found, although there has been a considerable amount of research. However, it has not proved to be an **NP**-complete problem [18]. SAT is a well known **NP**-complete problem. The question in SAT is to find satisfying valuation for some propositional logic formula in conjunctive normal form. Many other problems may be reduced to a SAT-instance and solved as such. For instance, planning and verification problems have been successfully solved as SAT-instances taking advantage of the effort put to creating efficient solvers.

2.1 Boolean Circuits

We treat Boolean circuits resembling [13]. A Boolean circuit is an acyclic directed graph. Each node in the graph is either an input node, or has associated a function. The in degree of node is the amount of the incoming arcs. Each input node has in degree 0. Each node with associated function has the in degree ≥ 1. An *output node* is a node with out degree 0.

We formalize this as follows. An acyclic graph with an associated function forms a Boolean circuit $C = (V, E, f)$, where the set of nodes is V, the set of edges is $E \in V \times V$, and nodes are mapped to different boolean functions using $f : V \rightarrow \{\wedge, \vee, \oplus, \neg, \top, \bot, x_1, x_2, x_3, ...\}$. The circuit satisfies following conditions.

$$\begin{aligned} \forall v \in V f(v) \in \{x_1, x_2, x_3, ...\} &\quad \text{implies} \quad indegree(v) = 0 \\ \forall v \in V f(v) \in \{\top, \bot\} &\quad \text{implies} \quad indegree(v) \in \{0, 1\} \end{aligned}$$

$$\forall v \in V f(v) \in \{\neg\} \quad \text{implies} \quad indegree(v) = 1$$
$$\forall v \in V f(v) \in \{\wedge, \vee, \oplus\} \quad \text{implies} \quad indegree(v) \geq 2$$

A path is a sequence of vertices $v_1, v_2, v_3, ...v_n \in V$ $(2 \leq n \leq |E|)$ such that $\forall i \in [1, n-1](v_i, v_{i+1}) \in E$. In the Boolean circuits there exists no path for which $v_1 = v_n$.

Truth assignment T for Boolean circuit is a function $V \to \{\top, \bot\}$ satisfying the following conditions. Given vertex v, let the set of predecessors be $p(v) = \{v\prime | (v\prime, v) \in E\}$, the set of true valued predecessors $tp(v) = \{v\prime | v\prime \in p(v), T(v\prime) = \top\}$ and the set of false valued predecessors $fp(v) = \{v\prime | v\prime \in p(v)$ and $T(v\prime) = \bot\}$. For all $v \in V$ following statements hold:

$$
\begin{aligned}
f(v) \in \{\top, \bot\} &\quad \text{implies} \quad (\forall v\prime \in p(v) : T(v\prime) = T(v))^{1} \\
f(v) = \top &\quad \text{implies} \quad T(v) = \top \\
f(v) = \bot &\quad \text{implies} \quad T(v) = \bot \\
f(v) = \wedge &\quad \text{implies} \quad (\exists v\prime \in p(v) \text{ such that } T(v\prime) = \bot \leftrightarrow T(v) = \bot) \\
f(v) = \vee &\quad \text{implies} \quad (\exists v\prime \in p(v) \text{ such that } T(v\prime) = \top \leftrightarrow T(v) = \top) \\
f(v) = \oplus &\quad \text{implies} \quad (|tp(v)| \text{ is odd } \leftrightarrow T(v) = \top)
\end{aligned}
$$

A decision problem CIRCUIT SAT is a problem where the solution is yes, whenever there exists a truth assignment for the circuit (possibly including an output given as a constraint) and no when it does not. CIRCUIT SAT is a **NP**-complete problem [18]. It has the corresponding function problem, where the solution is the satisfying truth assignment.

2.2 About Solvers for the SAT problem

Solvers used are either complete or incomplete. A complete method is is always able to find a solution, if such exists, and therefore also able to prove unsatisfiability. Any method, which does not have such a capability, is incomplete. Most solvers for SAT are based on DPLL-method [9, 8] or local search [21]. DPLL-method is a simple and complete method. The performance of DPLL-based solvers varies due to differences in heuristics, learning, backjumping and Boolean constraint propagation implementations. Local search based solvers are incomplete and based on the idea of starting with a solution candidate and gradually improving the candidate towards correct solution [20].

2.3 Factoring as a SAT Instance

We generate challenging SAT instances from factoring problem by implementing a multiplication circuit and assign the number to be factored as the output. The

[1] These acts as constraints.

truth assignment for such a Boolean circuit includes the factors.

A fundamentally different approach would be a circuit where the logic would be designed starting from the output and producing the inputs. However, we feel that implementing such a circuit is not as straightforward as our approach. Therefore it is left for the future research. Of course, there are also the combinations of these two approaches.

2.4 Notation for Compositional Circuits

The multiplier circuits are modular, and we need a notation for describing compositional circuits. A *compositional circuit* is a Boolean circuit defined by a set of Boolean circuits and set of connections between them. Compositional circuit $C = (M, E)$ is composed of a set Boolean circuits, M, called *modules* and a set of edges E between input and output nodes of the modules.

Node sets of each circuit are assumed to be distinct. The node set of a Boolean circuit corresponding the compositional circuit is formed by taking union of node sets of components. The input nodes of modules, which appear in E, are removed. The edge set contains all the edges of the modules, except the edges made redundant by connections between modules, i.e. if some input node in module 1 is connected to a output node in module 2, then each edge connecting the input node to the rest of the module is replaced by an edge starting from the output node. The mapping giving the associated functions to nodes is the union of the all mappings, except that nonexistent nodes (input nodes, which are connected) are removed.

Formally: A compositional circuit $C = (M, E)$, where $M = \{(V_1, E_1, F_1,),$ $..., (V_n, E_n, F_n)\}$ with preconditions

(i) $\bigcap_{i=1}^{i=n} V_i = \emptyset$ and

(ii) $(v, v') \in E$ implies $\text{indegree}(v) = 0$

defines the Boolean circuit $B_C = (V', E', F')$, where

$$\begin{aligned}
V' &= \{v | \exists i : v \in V_i \wedge \forall v' \in \bigcup_{i=1}^{i=n} V_i : (v', v) \notin E\} \\
E' &= \{(v, v') | (v, v') \in \bigcup_{i=1}^{n} E_i \wedge \forall v'' \in \bigcup_{i=1}^{i=n} V_i : (v'', v) \notin E\} \cup \\
&\quad \{(v', v'') | \exists v \in \bigcup_{i=1}^{i=n} V_i : (v', v) \in E \wedge (v, v'') \in \bigcup_{i=1}^{n} E_i\} \\
F'(x) &= F_i(x), \text{when } x \in V_i \text{ and not } \exists v \in V' : (v, x) \in E
\end{aligned}$$

If M is a module, then by M^a we denote the input or the output node a in this module. To denote the connection between the module M_1 input a and the module M_2 output b, we use pair (M_1^a, M_2^b).

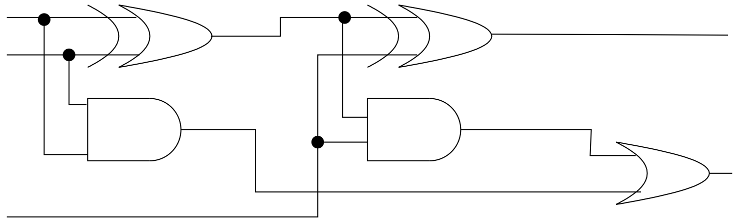

Figure 2: Full Adder. The gates correspond to full adder nodes as follows (from top left to right bottom): 1,2,sum,3,carryout. The incoming arcs from left are (top to bottom): in1, in2 and in3.

3 Multiplier Circuits

Two multiplier circuit designs have been implemented. These are a Braun multiplier [6], which consists of grid of full-adders processing the results from and-ports. The other is an adder tree [6], which uses adders to sum up all the partial products in tree fashion.

In the following a and b refer to multiplicands. With and $A(i)$ and $B(i)$ we refer to particular bits in their binary representation. The least significant bit has the index 1.

3.1 Modules for Circuits

Definition 1 *Full adder -module is defined as* $FA = (\{1, 2, 3, sum, carryout, in1, in2, in3\}, E, f)$*, where* $E = \{(in1, 1), (in2, 1), (in1, 2), (in2, 2), (1, 3), (in3, 3), (1, sum), (in3, sum), (2, carrout), (4, carryout)\}$ *and* $f = \{(1, \oplus), (sum, \oplus), (2, \wedge), (3, \wedge), (carryout, \vee), (in1, x1), (in2, x2), (in3, x3)\}$*. It adds the three inputs and outputs the result. The full adder -module is graphically drawn in Fig. 2.*

Definition 2 *And-module is defined as* $A = (\{out, in1, in2\}, E, f)$*, where* $E = \{(in1, out), (in2, out)\}$ *and* $f = \{(out, \wedge), (in1, x1), (in2, x2)\}$*. This definition is for notational consistency only and confusing. Note that this is just an and-gate, but we call it "and-module" to keep the notation consistent!*

3.1.1 Braun Multiplier

A Braun multiplier is a simple form of a parallel adder. All products of two bits $(A(i) \cdot B(j))$ are computed in parallel. The results are combined through an array of full adders. Figure 3 is a diagram of a 3-bit Braun multiplier. It multiplies two three-bit inputs $< A(3), A(2), A(1) >$ and $< B(3), B(2), B(1) >$ (therefore $A(1)$ and $B(1)$ being the least significant bits) to form six-bit output $< P(6), P(5), P(4), P(3), P(2), P(1) >$. Maximum delay from inputs $B(0)$ and

$A(n-1)$ to $P(2n-2)$ is $2n-2$ full adders and one and-gate. The delay scales as $O(n)$. A braun multiplier consists of n^2 and-gates and n^2-n full adders. Each full adder has 5 gates. Therefore Braun multiplier has $n^2+5(n^2-n)=6n^2-5n$ gates. This scales asymptotically as $O(n^2)$.

Braun multiplier may be also described formally. Let $F_n=\{F(1,1),...,F(1,n-1),...,F(n,1),...,F(n,n-1)\}$ where each element is isomorphic with full adder -module and node distinct. Let $A_n=\{A(1,1),...,A(1,n),...,A(n,1),...,A(n,n)\}$. Now we define n-bit Braun multiplier is compositional circuit, where set of modules is $M=A_n\cup F_n$ and the set of connections E between them defined as follows:

$$\begin{aligned}
E_1 &= \{(A(x,1),F(x-1,1)_{in3})|2\leq x\leq n-1\}\\
E_2 &= \{(A(n,y),F(n-1,y)_{in3})|1\leq y\leq n\}\\
E_3 &= \{(A(x,y),F(x,y-1)_{in1})|2\leq y\leq n,1\leq x\leq n-1\}\\
E_4 &= \{(F(x,y)_{carryout},F(x,y+1)_{in2})|1\leq y\leq n-1,1\leq x\leq n\}\\
E_5 &= \{(F(x,y)_{out},F(x-1,y+1)_{in1}|1\leq y\leq n-1,2\leq x\leq n-1\}\\
E_6 &= \{(F(x,n)_{carryout},F(x+1,n)_{in1}|1\leq x\leq n-2\}\\
E_7 &= \{(F(x,n)_{out},P(x+n))|1\leq x\leq n-1\}\\
E_8 &= \{(F(1,y)_{out},P(y+1))|1\leq y\leq n-1\}\\
E_9 &= \{F(n-1,n)_{carryout},P(2n))\}\\
E_{10} &= \{(A(x),A(x,y))|1\leq y,x\leq n\}\\
E_{11} &= \{(B(y),A(x,y))|1\leq y,x\leq n\}\\
E_{12} &= \{(A(1,1),P(1)\}\\
E &= \bigcup_{i=0}^{13}E_i
\end{aligned}$$

3.1.2 Carry Lookahead Adder

A carry lookahead adder may be used as a component of an adder tree multiplier [11]. This adder design is selected to enable fast carry propagation [15]. The design is composed of partial full adders and carry propagation logic. Carry propagation logic blocks can be connected as a tree, and therefore the design is easily adapted to any width of operands. Figure 6 describes the overall design. A partial full adder is described by Figure 4.

Definition 3 *A partial full adder-module is defined as* $PFA=(\{s,g,p,c,b,a\},E,f)$*, where* $E=\{(b,p),(a,p),(p,s),(c,s),(a,g),(b,g)\}$ *and* $f=\{(p,\oplus),(s,\oplus)$ $(g,\wedge),(a,x1),(b,x2),(c,x3)\}$.

A partial full adder produces three signals s(um), p(ropagate) and g(enerate). Inputs are c(arry), a and b. The sum is the sum of inputs. The generate signal corresponds to carry generation. A carry is generated when both a and

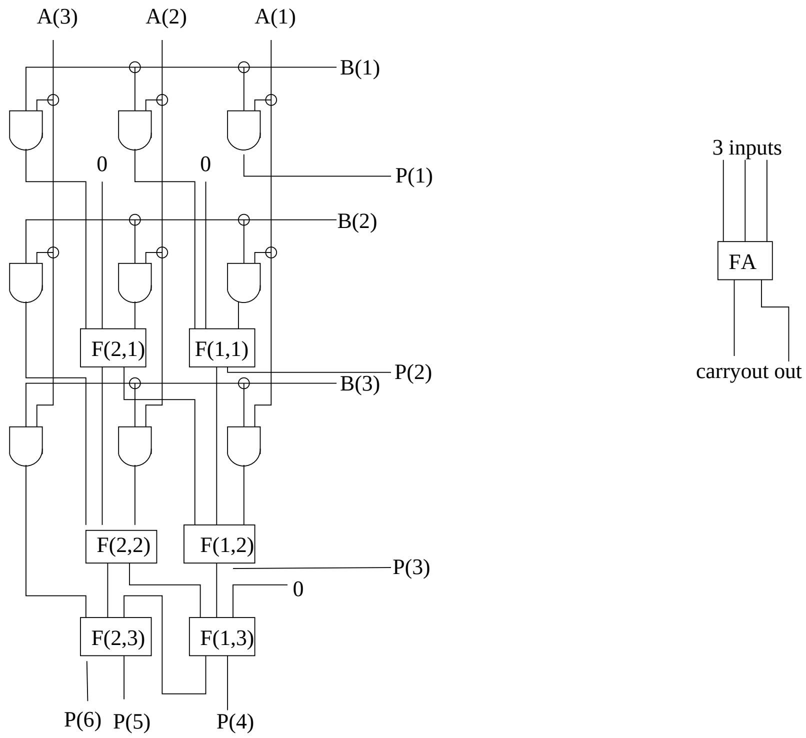

Figure 3: 3-bit Braun multiplier. The And-modules are indexed in the same fashion as the full adders, for example A(1,1) is the And-module on top right.

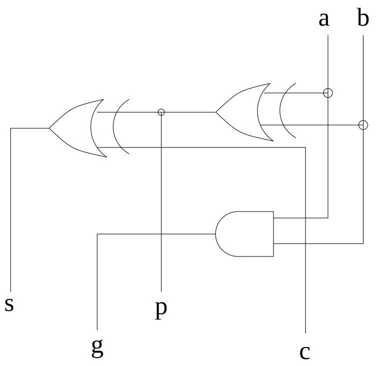

Figure 4: Partial full adder

b are true. Carry propagation means that the previously generated carry is propagated forwards. This is indicated by generating propagate signal. The propagate signal is XOR of a and b.

The carry propagation block contains logic for carry generation. It also has logic to provide generate and propagate signals to the higher level carry propagation block. Each carry is generated by checking whether some carry is propagated to this position. This is the case, if the previous position has the generate signal set. Another possibility is that some position generates the signal and all positions between it and this one have the propagate signal set. In the experiments, propagation block of width 4 is used. The final sum is produced by the partial full adders and carry progation logic just produces the carries required by the partial full adders. The advantage of the design is that none of the partial full adder outputs connected to the carry progation logic depends on the carry produced by the carry propagation logic.

Propagation block is defined as follows. Let the incoming signals be P_0, P_1, P_2, P_3, G_0, G_1, G_2, G_3 and C_0 (four propagate signals, four generate signals and carry respectively). The block outputs following signals C_1, C_2, C_3, P_{0-3}, G_{0-3} (three carries and higher level propagate and generate signals respectively). These defined as follows

$$\begin{aligned}
C_1 &= (C_0 \wedge P_0) \vee G_0 \\
C_2 &= (C_0 \wedge P_0 \wedge P_1) \vee (G_0 \wedge P_1) \vee G_1 \\
C_3 &= (C_0 \wedge P_0 \wedge P_1 \wedge P_2) \vee (G_0 \wedge P_1 \wedge P_2) \vee (G_1 \wedge P_2) \vee G_2 \\
\text{Propagate} &= P_0 \wedge P_1 \wedge P_2 \wedge P_3 \\
\text{Generate} &= (C_0 \wedge P_0 \wedge P_1 \wedge P_2 \wedge P_3) \vee (G_1 \wedge P_2 \wedge P_3) \vee (G_2 \wedge P_3) \vee G_3
\end{aligned}$$

Definition 4 *Carry propagation block-module is defined as* $CPB =$ *({c0, c1, c2, c3, p0, p1, p2, p3, g0, g1, g2, g3, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, gener-*

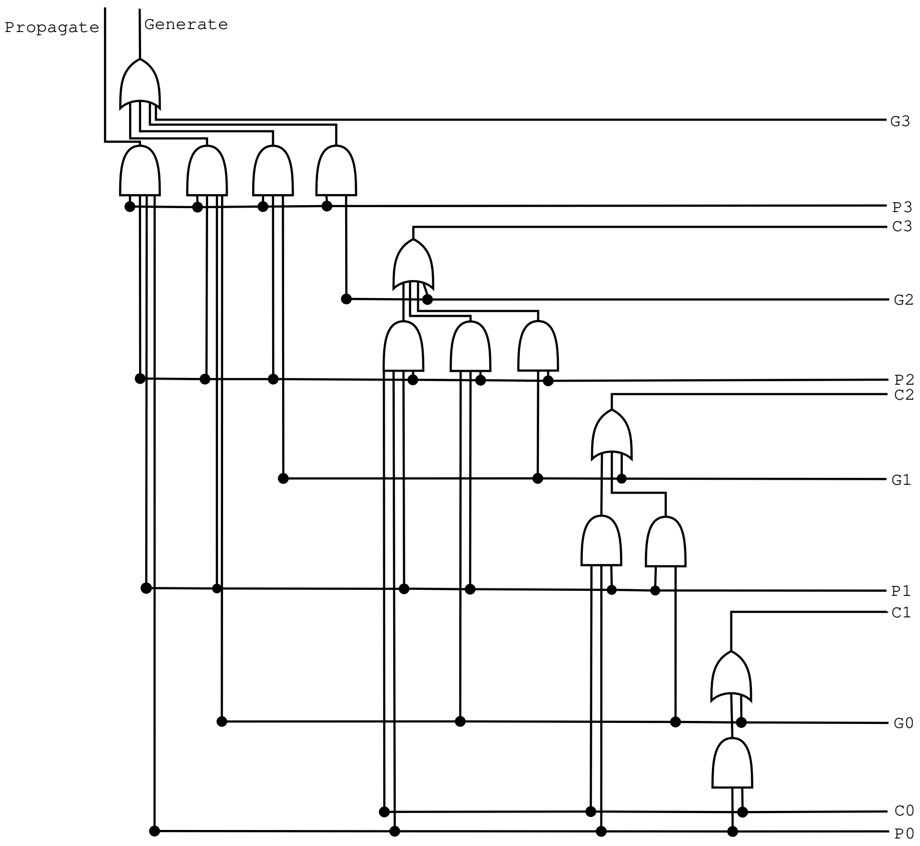

Figure 5: Carry propagation logic block

ate, propagate}, E, f), where E = {(c0,a0), (p0,a0), (a0,c1), (g0,c1), (c0,a1), (p0,a1), (p1,a1), (g0,a2), (p1,a2), (a1,c2), (a2,c2), (g1,c2), (c0,a3), (p0,a3), (p1,a3), (p2,a3), (g0,a4), (p1,a4), (p2,a4), (g1,a5), (p2,a5), (g3,c3), (a3,c3), (a4,c3), (a5,c3), (p0,propagate), (p1,propagate), (p2, propagate), (p3, propagate), (c0,a6), (p0,a6), (p1,a6), (p2,a6), (p3,a6), (g1,a7), (p2,a7), (p3,a7), (g2,a8), (p3,a8), (a6,generate), (a7,generate), (a8,generate), (g3,generate)} and f = {(a0, ∧), (a1, ∧), (a2, ∧), (a3,∧), (a4,∧), (a5,∧), (a6,∧), (a7,∧), (a8,∧), (c1, ∨), (c2, ∨), (c3, ∨), (generate, ∨), (propagate, ∧), (c0,x0), (p0,x1), (p1,x2), (p2,x3), (p3,x4), (g0,x5), (g1,x6), (g2,x7), (g3,x8)}. This is also drawn in Fig. 5.

In a carry lookahead adder the first level of the propagation logic is connected to partial full adders. Subsequent levels are connected to propagation logic blocks of the previous level.

If the width of the inputs is not exactly 4^n (the carry propagation blocks are connected as a tree) for some $n \in Z$, only necessary carry propagation logic blocks are generated. The inputs, where no real propagate- and generate-signals exist, are set to false. The maximum propagation delay of carry lookahead adder of L levels is $4L+2$ according [15].

Definition 5 *Carry lookahead adder-module of width n is $CLA_n = (M, E, f)$, where M=$\{PFA_1, PFA_2, \ldots, PFA_{\ulcorner n/4 \urcorner}, CPL_{1,1}, CPL_{1,2}, \ldots, CPL_{1,\ulcorner n/4 \urcorner}, CPL_{2,1}, CPL_{2,2}, \ldots, CPL_{2,\ulcorner n/4^2 \urcorner}, \ldots, CPL_{1,\log_4 n}, F\}$ where all modules are distinct and each items named like PFA are isomorphic with PFA-module and each item named like CPL is isomorphic with CPL module. F is a false module, consisting of one gate v, for which $f(v) = \perp$. The connection set is defined as follows $E = \bigcup_{i=1}^{i=7} E_i$, where*

$$
\begin{aligned}
E_1 &= \quad (PFA_k^p, CPL_{1,\ulcorner k/4 \urcorner}^{p(k mod 4)}), k \in \{1, .., n\} \\
E_2 &= \quad (PFA_k^g, CPL_{1,\ulcorner k/4 \urcorner}^{g(k mod 4)}), k \in \{1, .., n\} \\
E_3 &= \quad (CPL_{i-1,k}^{propagate}, CPL_{i,\ulcorner k/4 \urcorner}^{p(k mod 4)}), k \in \{1, .., n\}, i \in \{1, .., \frac{n}{4^i}\} \\
E_4 &= \quad (CPL_{i-1,k}^{generate}, CPL_{i,\ulcorner k/4 \urcorner}^{g(k mod 4)}), k \in \{1, .., n\}, i \in \{2, .., \frac{n}{4^i}\} \\
E_5 &= \quad (CPL_{1,\ulcorner k/4 \urcorner}^{c(k mod 4)}, PFA_k^c), k \in \{1, .., n\} \\
E_6 &= \quad (CPL_{i+1,\ulcorner k/4 \urcorner}^{c(k mod 4)}, CPL_{i,k}^{c0}), k \in \{1, .., \frac{n}{4^i}\}, i \in \{1, .., \log_4 n\} \\
E_7 &= \quad \{(F^f, CPL_{j,k}^i) | \textit{where no edge exists in} \bigcup_{l=1}^{i=6} E_i \textit{ for this CPL-input } i\}
\end{aligned}
$$

In further discussion the inputs of a carry lookahead adder block are named as $A(i)$ and $B(i)$ corresponding the bits of the input numbers.

3.1.3 The Adder Tree Multiplier

The adder tree multiplier sums up the partial products of the form $B(i) \cdot A$. From these, we can form the product $< B(i+1), B(i) > \cdot A$ by shifting $B(i+1)$ by one bit and adding results together. In the same fashion we may continue to combine two-bit partial products to four-bit $(< B(i+3), B(i+2), B(i+1), B(i) > \cdot A)$. Finally we get the actual product $A \cdot B$. The multiplier formed in this way is formed as a tree. For reference, a high level picture of a four bit adder tree multiplier is shown in Fig. 7. In this implementation we use carry lookahead adders.

When considering increasing the input length, total number of adders grows linearly, and the adders also have to be made linearly wider, to accumulate larger inputs. The depth of an adder tree grows logarithmically, as does the depth of an invidual adder as it is made linearly wider.

The propagation delay can be approximated as follows assuming that we are multiplying n-bit numbers. There are $\log_2 n$ (rounded upwards to integral number) adders on any path from input to output. In a carry lookahead adder

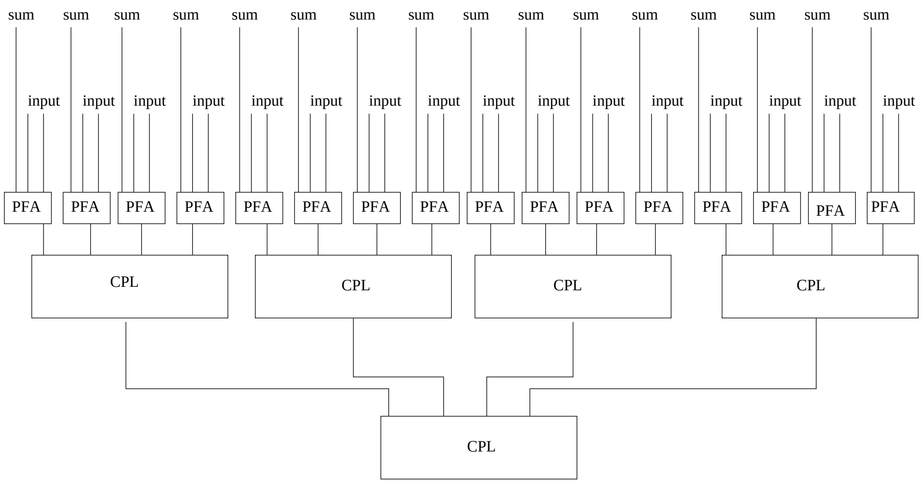

Figure 6: Carry lookahead-adder overview. Lines represent signals between the blocks.

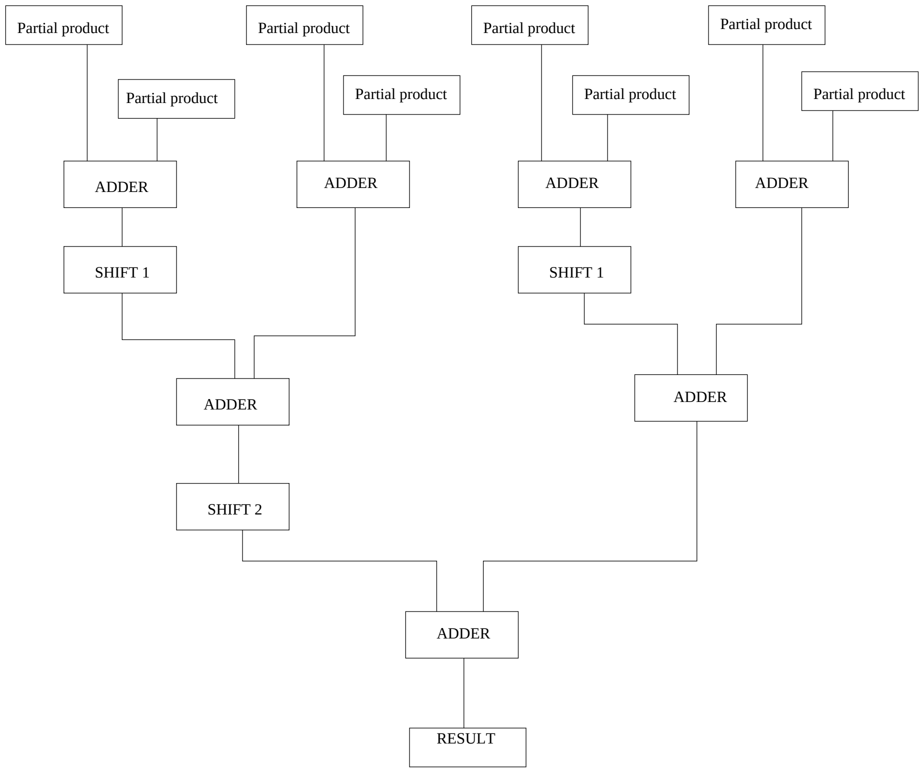

Figure 7: 4-bit adder tree multiplier

of the length m there is a maximum delay (from the first input bit to the last output bit) of $4\log_4 m+2$ gates. Now, the maximum adder width in the current implementation is $3n$ bits. Approximating upwards all adders to this width, one may calculate that there is $\log_2 n(4\log_4 3n+2)$ gates. Therefore propagation delay is $O(\log(n\log n))$.

Making the same assumptions and approximations, the amount of gates can be calculated as follows. There are $n-1$ adders in the circuit. A carry lookahead adder of the length $3n$ has $\frac{3n}{2}-1$ propagation logic blocks. Each propagation logic block has a constant (14) number of gates. Additionally an adder has $3n$ partial full adders, which have constant number (3) of gates. In total there are $(\frac{3n}{2}-1)*14+3n*5$ gates in each adder. Therefore the multiplication circuit has $(n-1)*((\frac{3n}{2}-1)*14+3n*5)$ gates, which grows as $O(n^2)$.

Definition 6 *An adder tree multiplier of width n is the compositional circuit constructed by Algorithm 3.1. In M the CLA-modules are isomorphic with the carry lookahead adder module and And-modules are isomorphic with the and-module.*

Algorithm 3.1: ADDERTREEMULTIPLIER($width$)

$M \leftarrow \{And_{i,j} | 1 < i,j < width\}$
$M \leftarrow M \cup \{CLA_{1,i} | 1 < i < width/2\}$
$E \leftarrow \{(CLA_{1,i}^{a_j}, And_{2*i,j}^{out}, CLA_{1,i}^{a_j}, And_{2*i,j}^{out}, CLA_{1,i}^{b_j}\}$
$(M,E) \leftarrow$ CONSTRUCTLEVEL($M, E, \emptyset, width, width/4, 1$)
$Connectunconnected CLA - inputstoFalseinE$
return $((M,E))$

Algorithm 3.2: CONSTRUCTLEVEL($M, E, UnconnectedAdders, origwidth, width, level$)

$Adders \leftarrow \{CLA_{1,level}, ...CLAwidth, level\}$
$Connections \leftarrow \{(CLA^{a_{j+2^{level-1}}}_{i,level}, CLA^{o_j}_{i*2,level-1}), (CLA^{b_j}_{i,level}, CLA^{o_j}_{i*2+1,level-1})|$
$1 < i < width, 1 < j < origwidth + level - 1\}$
$M = M \cup Adders$
$E = E \cup Connections$
if $|UnconnectedAdders| = 2$

$$\textbf{then} \begin{cases} width \leftarrow width + 1 \\ M = M \cup \{CLA_{width,level}\} \\ UCLA_1 \leftarrow UnconnectedAdders[1] \\ UCLA_2 \leftarrow UnconnectedAdders[2] \\ offset \leftarrow 2^{\text{level of } UCLA_1} \\ E = E \cup \{(CLA^{a_{j+offset}}_{width,level}, UCLA^{o_j}_1), (CLA^{b_j}_{width,level}, UCLA^{o_j}_1)| \\ 1 < i < width, 1 < j < origwidth + level - 1\} \end{cases}$$

if width is odd
  then $UnconnectedAdders = UnconnectedAdders \cup \{CLA_{level,width}\}$
if $width > 1$
  then return (CONSTRUCTLEVEL($M, E, UnconnectedAdders, origwidth, width/2, level + 1$))
  else return ((M, E))

4 Description of the Experiments

4.1 Number Classes Used in the Experiments

In the experiments we factored different kinds of numbers. We used composites, even numbers, primes and random numbers. A $2n$ bits wide composite number is constructed by generating two primes between 2^n and 2^{n+1} and multiplying. Therefore, the generated composite numbers are products of two primes about half of the product size. Even n bit wide numbers are generated by generating random number $n-1$ bits and multiplying it by two. Primes are generated with a function for prime generation from the GMP-library [10] and random numbers are also generated with a function from the same GMP-package. The GMP version employed uses trial divisions and the Rabin-Miller test to verify that primes are indeed primes. Rabin-Miller is a probabilistic method for checking whether some number is a prime or not. Very pessimistic probability for a false prime passing the test is $\frac{1}{16}$ with two repetitions used. In the reality the probability is actually much smaller [19].

In the experiments we varied the amount of factors in the number. Often it is believed that problems get easier, as they have more solutions. A CIRCUIT SATISFIABILITY -instance consisting of a multiplication circuit and a composite number having n different factors has 2^n solutions. This can be proven using induction as follows. First of all, 1 has zero factors, and has only 1 solution $1 = 1 \cdot 1$. Induction hypothesis: presume that x has n different prime factors and 2^n solutions. Induction step: $x \cdot y$ (y is a prime) has 2^{n+1} solutions, because for any solution n, p we can find two new unique solutions $n \cdot y, p$ and $n, p \cdot y$. These solutions obviously are solutions. They must be also unique, because otherwise the two original solutions were the same solution. This result is a bit odd, because conventionally we take account the commutativity of multiplication to consider n, p and p, n as the same solution. However, from the solver point of view, this is not the case.

Optimizations on the circuit also reduce the amount of solutions. If one of the inputs must be at most square root of the product, then the solution, where the constrained input is larger than the square root of product, is eliminated.

In the results of the experiments you see that we have run experiments with varying number of solutions. The instances with no solutions are based on prime numbers with factor 1 disabled. The instances with 2 solutions are based on composite numbers with factor 1 disabled. The instances with 4 factors are composite instances with factor 1 enabled. Sometimes the instances with supposedly 2 or 4 solutions have only 1 or 3 solutions, as it happens that factors of the composite are the same number.

The rest of the instances have 2^i solutions where $i \in \{3, 4, 5, 6, 7, 8, 9\}$. These have been generated for the width n by generating a list of primes starting from 2 and containing all the primes in ascending order. Then the program forms numbers by multiplying i distinct numbers in list (starting from the beginning of the list). Once a number with the specified width is formed, it is used in a problem instance, with seed 0. If a greater seed is requested, more numbers are formed until we have formed seed+1 numbers. In the case that result of the multiplication is too large number the generation is aborted.

When we have successfully generated number of the specified width, then the corresponding SAT instance has 2^i solutions from the solver perspective. Forming the numbers in this way has disadvantage of large amounts of memory and CPU time being used for large numbers. The algorithm is neither guaranteed to produce all the numbers fulfilling the criterion, however it is able to generate sufficient amount of the purpose of creating factoring instances.

4.2 Software and Hardware

We factored the numbers using several different solvers. The solvers for Boolean circuit satisfiability and propositional logic satisfiability (SAT) were used. SAT-solvers Satz [16], Chaff [17], Sato [23] and Relsat [1] are complete solvers based on DPLL-method [9, 8]. Bcsat [13] is complete solver for CIRCUIT SATISFIABILITY. Versions used are Satz 2.14, Bcsat v0.3, Z-Chaff Z2001.2.17 and relsat 2.00.

The tests were run on machine based on Pentium II processor running at 450MHz and having 256MB of memory. As the operating system we used Debian GNU/Linux 2.1.

We have implemented a tool for generating factoring problem instances using the multiplier circuits presented in Section 3. It is available through WWW at http://www.tcs.hut.fi/%7etpyhala/. See Appendix A for usage instructions. The generator natively generates problem instances in Bcsat and Smodels [22] formats. Bcsat format documented at [12] is suitable for specifying a Boolean circuit represented in Section 2. Smodels format specifies a stable model logic program. A Boolean circuit can be easily mapped to such a logic program.

However, most SAT solvers require that their input is given conjuctive normal form (CNF) in dimacs format. Bcsat instance may be translated into dimacs format with the Bc2cnf-tool [12]. Conversion is based on mapping each gate locally to CNF. For example, consider circuit with one input gate connected to a not gate, where $V = \{v, v'\}$, $E = \{(v, v')\}$ and $f = \{(v, x_1), (v', \neg)\}$. This can be transformed to CNF form by a local transformation resulting the following formula $(v \vee v') \wedge (\neg v \vee \neg v')$. Other gates can be transformed locally also, for further information see [13].

Our generator consists of several small programs and a front end where respective output format, i.e. dimacs, bcsat and smodels, can be selected easily. At the time of running the tests the bc2cnf tool was not available, but we used Bcsat with option -noreduce instead.

4.3 Restrictions for Answers

The circuits used are designed to multiply two m-bit numbers and producing $2m$-bit output at maximum. We are using the same design to operate backwards. Consider our problem of factoring. We have a number n, which is has length m when represented in binary number. We are searching for solution $p \cdot q = n$.

First interesting property is that both p and q cannot be larger than $\sqrt{n}$. This can be easily proved by assuming the opposite, which leads to $n = p \cdot q > \sqrt{n}^2 = n$ – a contradiction.

Second interesting property is that p or q might be larger than $\frac{m}{2}$ bits when represented in binary. For example, $15_{10} = 1111_2$, and one solution is $3_{10} \cdot 5_{10}$. When represented in binary these numbers are $5_{10} = 0101_2$ and $3_{10} = 0011_2$, of which the first is larger than 2 bits.

The conclusion is that when factoring a m bit number we need a circuit with input size of m bits, which with these circuits makes the output $2m$ bit wide. Therefore, the upper half of the output is zeroed. By the first property we introduce a trick to reduce search space by setting upper half of the other input to zero.

4.3.1 Reduced Braun Multiplier

The Braun multiplier design has two n-bit inputs and $2n$-bit outputs as explained in previous paragraph. From optimization point of view we can take further the approach of setting upper half of the multiplier output to zero. We may take away the gates and connections which are made constant by this assignment. With a Braun multiplier this approach succeeds to eliminate about half of the full adders.

This could be refined even further by removing also the constant gates introduced by assigning upper half of other of the inputs zero. This eliminates about half of the and-gates and one fourth of the remaining full adders. However, from the performance point of view neither optimization should be important. Determination of the truth assignment for these parts of the circuit should be straightforward and not require any search.

5 Results

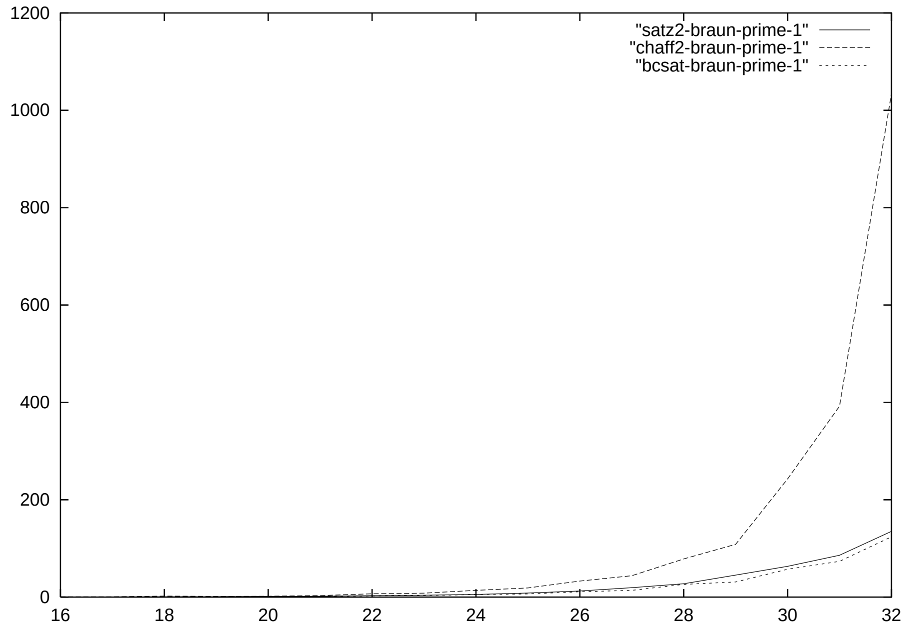

Figure 8: Braun multiplier, prime numbers (CPU seconds used on y-axis and number width in bits on x-axis)

5.1 Solvers compared

We measured the time required to solve different kinds of factoring problems with different solvers. The test setup was simply to run the solver and measure time using Linux standard "time"-tool measuring CPU usage of the process. The results for different solvers with the composite numbers are represented in Figures 9 and 11 and for the primes in Figures 8 and 10. Complete results including the cases where number of factors was varied are shown in Tables 1,2, 3 and 4.

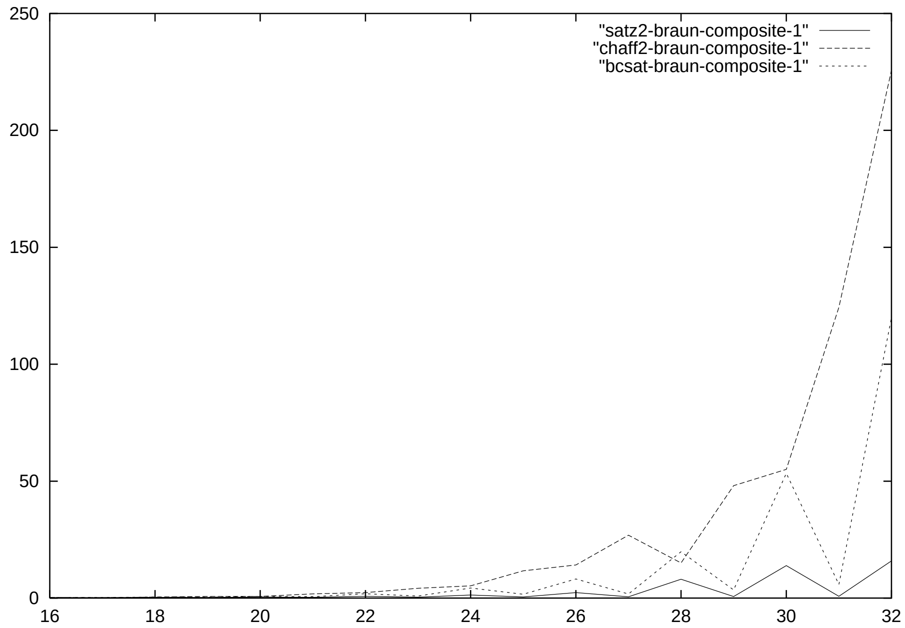

Figure 9: Braun multiplier, composite numbers (CPU seconds used on y-axis and number width in bits on x-axis)

5.2 Quest for the hard problem

Results in Tables 1, 2, 3 and 4 show that the problem gets harder as the amount of the solutions to the SAT instance is reduced and vice versa. This seen by the required CPU-time decreasing (to the right) as the number of solutions increasing (to the right). Surprisingly, this transition from hard to easier problems isn't smooth at all. Similar behavior is observed with several solvers and therefore the behavior is not solver dependant. However,the trend seems to be that the problems with more solutions are easier.

Also when comparing instances with no solution in Figures 8 and 10 to the instances with two solutions in Figures 11 and 9 we observe that instances with no solution are harder than instances with two solutions. When solving instance with no solution, complete solver effectively proves that the number is prime, because there exists no solutions.

Dependence of width is obvious too and - at least with hard problems - seems to be exponential. The time required to solve a problem increases exponentially when compared to width of the numbers and multiplication circuit.

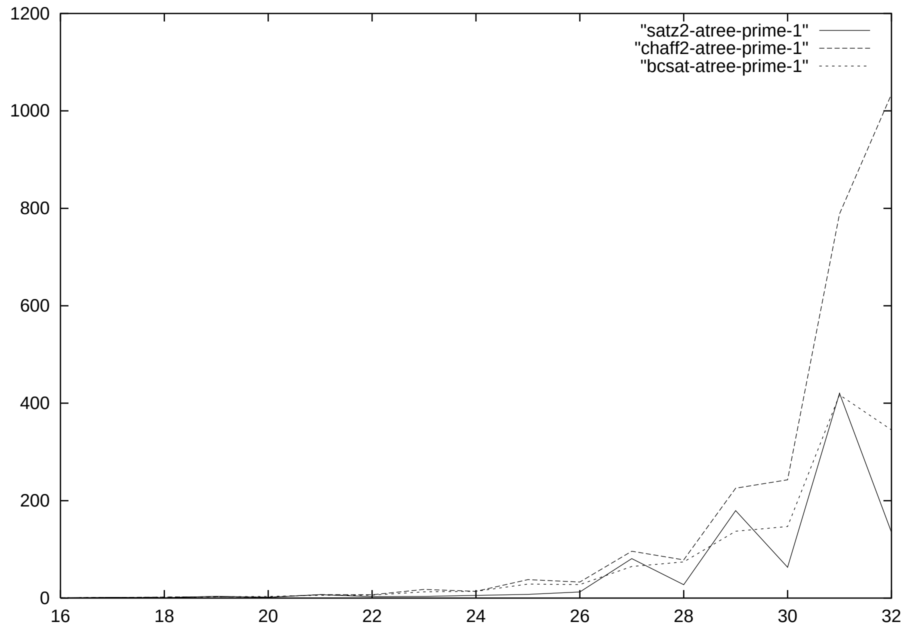

Figure 10: Atree multiplier, prime numbers (CPU seconds used on y-axis and number width in bits on x-axis)

6 Conclusions

Satz seems to be the most efficient solver for the problems with the encodings and parameters used in these experiments. The amount of time required to solve given problem seems to increase if the width of the circuit is increased or if the amount solutions is decreased. The increase w.r.t. increased circuit width is exponential, and therefore seems to support hypothesis that factoring is a hard, not polynomially solvable problem, when considering the hard instances. These are also those, which are used in cryptography.

We are also able to conclude that we can efficiently solve only factoring instances, which are pretty small. The number theoretic methods are much more efficient, when solving these factoring instances.

The generator created is available through WWW at address http://www.tcs.hut.fi/%7etpyhala/. Interested readers are encouraged to use it to generate SAT-instances! See appendix A for usage instructions.

References

[1] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth*

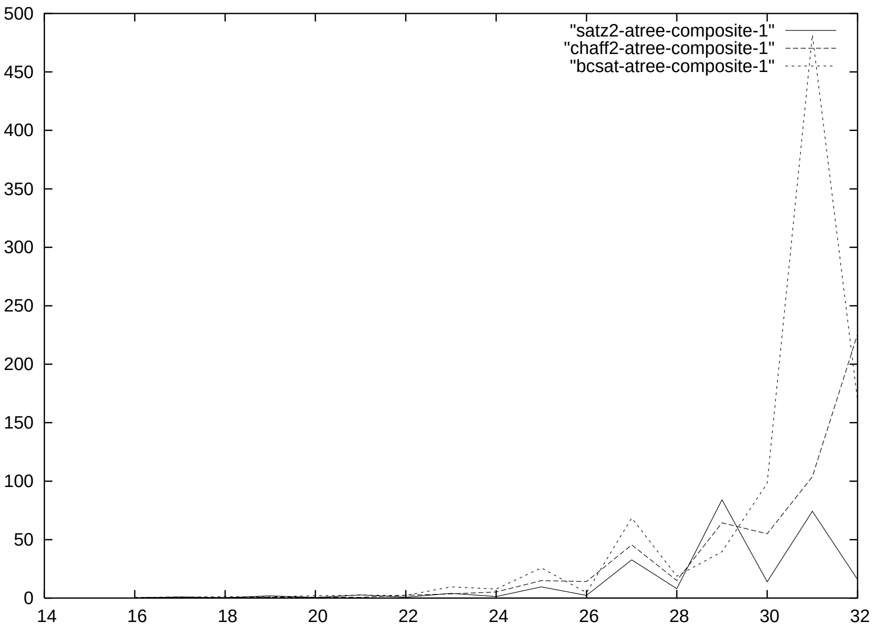

Figure 11: Atree multiplier, composite numbers (CPU seconds used on y-axis and number width in bits on x-axis)

National Conference on Artificial Intelligence (AAAI'97), pages 203–208, Providence, Rhode Island, 1997.

[2] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *11th International Conference in Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC'99)*, pages 317–320. ACM, 1999.

[5] A. Borälv. The industrial success of verification tools based on stalmarck's method. In *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer, 1997.

width	0	2	4	8	16	32	64	128	256	512
16	0.36	0.21	0.20	0.23	0.20	0.17	0.02	0.00	0.00	0.00
17	0.48	0.22	0.20	0.35	0.25	0.23	0.00	0.00	0.00	0.00
18	0.76	0.31	0.28	0.47	0.30	0.24	0.00	0.00	0.00	0.00
19	0.90	0.27	0.27	0.87	0.39	0.33	0.30	0.00	0.00	0.00
20	1.45	0.43	0.31	1.04	0.39	0.36	0.32	0.00	0.00	0.00
21	1.87	0.33	0.31	2.01	0.58	0.67	0.42	0.00	0.00	0.00
22	2.86	0.78	0.69	2.31	0.49	0.86	0.43	0.40	0.00	0.00
23	3.80	0.41	0.40	4.43	0.86	2.07	0.69	0.46	0.00	0.00
24	5.60	1.30	1.25	5.31	0.81	2.55	0.76	0.58	0.00	0.00
25	8.39	0.48	0.47	10.34	1.08	6.81	1.00	0.74	0.00	0.00
26	12.44	2.33	2.24	11.71	1.34	7.75	1.07	0.87	0.69	0.00
27	19.44	0.57	0.56	23.86	1.77	19.64	1.97	1.79	1.11	0.00
28	27.46	8.06	8.50	26.02	3.27	21.90	2.45	1.27	1.02	0.00
29	45.09	0.66	0.65	52.41	1.99	44.76	4.06	2.35	1.38	0.00
30	63.34	13.89	13.63	65.48	7.50	53.27	3.08	2.79	1.84	0.00
31	85.95	0.76	0.75	123.78	5.79	110.17	4.24	8.17	2.89	1.88
32	135.16	15.91	15.19	142.10	19.08	122.07	4.96	15.49	4.33	2.31
33	0.00	7.61	-	-	-	-	-	-	-	-
34	284.23	221.03	-	-	-	-	-	-	-	-
35	519.26	1.01	-	-	-	-	-	-	-	-
36	633.00	183.54	-	-	-	-	-	-	-	-
37	1172.34	1.15	-	-	-	-	-	-	-	-
38	1415.93	336.02	-	-	-	-	-	-	-	-
39	2579.07	1.32	-	-	-	-	-	-	-	-
40	3152.25	1018.44	-	-	-	-	-	-	-	-

Table 1: Satz results for Braun multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard "time"-utility. We didn't run any tests, whose cells are marked with dash (-).

[6] Stephen Brown and Zvonko Vranesic. *Fundamentals of digital logic with VHDL design*. McGraw Hill, 2000.

[7] Stephen A. Cook and David G Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.

[8] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Communications of ACM*, 5(7):394–397, 1962.

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[10] http://www.swox.com/gmp/.

width	0	2	4	8	16	32	64	128	256	512
16	0.36	0.21	0.20	0.62	0.54	0.54	0.06	0.00	0.00	0.00
17	1.61	0.85	0.53	0.51	0.51	0.59	0.08	0.00	0.00	0.00
18	0.76	0.31	0.28	0.93	0.76	0.70	0.00	0.00	0.00	0.00
19	3.60	1.98	1.93	1.55	1.06	0.86	0.70	0.00	0.00	0.00
20	1.45	0.43	0.31	2.01	1.23	1.04	0.96	0.00	0.00	0.00
21	6.94	2.73	2.15	2.95	1.55	1.39	1.22	0.00	0.00	0.00
22	2.86	0.78	0.69	3.19	1.68	1.66	1.38	1.44	0.00	0.00
23	3.51	4.05	2.41	5.34	1.80	2.55	1.54	1.33	0.00	0.00
24	5.60	1.30	1.25	6.01	2.39	3.41	2.11	1.35	0.00	0.00
25	7.56	9.54	8.83	12.13	4.39	10.19	3.31	2.15	0.00	0.00
26	12.44	2.33	2.24	12.32	3.57	4.84	3.42	1.97	1.78	0.00
27	80.97	32.65	31.91	27.32	8.87	9.28	4.85	3.96	4.19	0.00
28	27.46	8.06	8.50	30.28	6.59	15.86	6.40	1.81	3.45	0.00
29	179.61	84.09	69.97	64.13	9.67	4.31	2.35	1.68	1.56	0.00
30	63.34	13.89	13.63	62.18	14.30	31.19	13.48	9.03	5.63	0.00
31	420.45	74.21	72.66	185.55	24.70	157.07	16.67	25.01	9.16	6.61
32	135.16	15.91	15.19	177.01	42.67	153.67	18.40	50.39	14.77	11.55

Table 2: Satz results for adder tree multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard "time"-utility.

[11] http://www.fpga-guru.com/multipli.htm#row adder tree multipliers.

[12] Tommi A. Junttila. bc2cnf translator from contrained boolean circuits into dimacs cnf. http://www.tcs.hut.fi/ tjunttil/circuits/.

[13] Tommi A. Junttila and Ilkka Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic – CL 2000; First International Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567, London, UK, July 2000. Springer, Berlin.

[14] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[15] Charles R. Kime and M. Morris Mano. *Logic and Computer Design Fundamentals.* Prentice Hall, 1997.

[16] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, 1997.

width	0	2	4	8	16	32	64	128	256	512
16	0.33	0.24	0.21	0.20	0.18	0.17	0.00	0.00	0.00	0.00
17	0.42	0.25	0.23	0.36	0.26	0.23	0.09	0.00	0.00	0.00
18	0.64	0.42	0.33	0.38	0.30	0.24	0.00	0.00	0.00	0.00
19	0.81	0.38	0.35	0.68	0.43	0.35	0.34	0.00	0.00	0.00
20	1.24	0.73	0.60	0.81	0.45	0.41	0.39	0.00	0.00	0.00
21	1.53	0.61	0.58	1.47	0.50	0.63	0.47	0.22	0.00	0.00
22	2.65	1.86	1.52	1.70	0.59	0.84	0.53	0.52	0.00	0.00
23	3.12	0.89	0.92	3.22	0.72	1.65	0.82	0.67	0.00	0.00
24	5.25	4.28	3.34	3.74	0.79	2.07	0.72	0.63	0.00	0.00
25	6.56	1.64	1.17	7.46	1.12	4.93	1.16	1.01	0.42	0.00
26	11.09	8.15	6.86	8.54	1.32	5.25	1.05	0.96	0.88	0.00
27	13.80	1.75	1.73	16.11	1.64	13.02	2.23	1.35	1.22	0.00
28	26.08	19.79	16.65	19.66	2.40	15.09	2.35	1.50	1.19	0.00
29	31.08	3.46	3.86	39.71	2.94	31.09	3.03	2.46	1.71	0.52
30	57.11	53.34	43.31	46.27	4.78	34.44	2.78	3.00	2.08	0.00
31	73.54	5.81	5.73	96.52	6.85	80.64	5.28	6.31	3.67	2.56
32	123.86	119.89	100.92	109.09	12.18	84.04	5.00	9.78	4.17	2.70

Table 3: Bcsat results for Braun multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard "time"-utility.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[18] Christos H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[19] Bruce Schneier. *Applied Cryptography (Second Edition).* John Wiley & Sons, 1996.

[20] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, pages 521–432, Providence RI, 1993.

[21] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 337–343. American Association for Artificial Intelligence, 1994.

[22] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

width	0	2	4	8	16	32	64	128	256	512
16	0.71	0.60	0.55	0.56	0.38	0.55	0.00	0.00	0.00	0.00
17	1.38	0.78	0.65	1.07	0.84	0.79	0.28	0.00	0.00	0.00
18	1.71	1.16	1.10	1.26	0.96	0.85	0.00	0.00	0.00	0.00
19	2.61	1.03	1.01	2.08	1.22	1.06	0.99	0.00	0.00	0.00
20	3.28	2.07	2.01	2.79	1.36	1.18	1.06	0.00	0.00	0.00
21	5.72	2.47	0.00	5.26	1.95	2.23	1.74	0.22	0.00	0.00
22	6.04	2.24	2.22	6.39	1.96	2.89	1.82	0.52	0.00	0.00
23	12.89	9.58	0.00	10.36	3.84	1.93	1.95	0.67	0.00	0.00
24	13.96	7.69	7.05	13.02	3.80	4.36	3.00	0.63	0.00	0.00
25	28.78	25.83	25.94	5.70	5.98	6.48	3.17	1.01	1.28	0.00
26	27.67	4.68	5.48	11.46	4.26	11.50	4.97	0.96	3.32	0.00
27	64.86	68.45	67.64	25.83	7.64	25.27	7.13	1.35	4.67	0.00
28	74.30	18.76	15.42	43.58	8.92	35.17	7.65	1.50	5.51	0.00
29	137.29	39.65	41.41	135.51	19.42	118.01	9.36	2.46	6.61	2.44
30	146.93	98.24	96.98	143.19	10.94	139.13	17.45	3.00	7.83	0.00
31	417.05	481.78	471.67	215.43	84.86	19.04	36.14	6.31	7.58	6.45
32	345.34	169.08	167.24	192.46	59.41	17.42	50.94	9.78	7.00	7.97

Table 4: Bcsat results for adder tree multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard "time"-utility.

[23] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97)*, volume 1249 of *LNAI*, pages 272–275, 1997.

width	0	2	4	8	16	32	64	128	256	512
16	0.78	0.28	0.42	0.09	0.07	0.07	0.00	0.00	0.00	0.00
17	0.90	0.18	0.31	0.47	0.21	0.16	0.00	0.00	0.00	0.00
18	2.04	0.44	0.42	0.49	0.38	0.26	0.00	0.00	0.00	0.00
19	1.56	0.72	0.75	0.41	0.35	0.13	0.10	0.00	0.00	0.00
20	1.86	0.72	1.30	0.58	0.45	0.24	0.19	0.00	0.00	0.00
21	3.25	1.80	3.10	1.14	0.68	0.70	0.62	0.00	0.00	0.00
22	6.67	2.36	2.38	0.92	0.63	0.54	0.46	0.48	0.00	0.00
23	8.04	4.18	3.75	1.95	0.91	0.69	0.42	0.37	0.00	0.00
24	13.89	5.22	5.16	2.66	1.08	0.93	0.76	0.85	0.00	0.00
25	18.92	11.67	11.03	5.19	2.02	1.60	1.23	1.00	0.00	0.00
26	32.87	14.16	10.49	5.38	2.55	1.92	1.24	0.99	1.49	0.00
27	43.98	26.90	19.14	10.03	3.83	2.87	1.31	0.85	0.51	0.00
28	78.30	15.00	13.61	10.91	3.26	2.80	1.55	1.08	0.92	0.00
29	108.46	48.06	23.26	29.33	11.48	6.95	4.60	2.10	1.30	0.00
30	242.79	55.04	48.76	23.88	9.92	6.97	4.20	2.62	1.49	0.00
31	392.29	124.35	75.96	60.89	26.15	15.38	5.67	4.53	2.31	2.01
32	1034.32	225.70	0.00	62.13	18.54	15.37	7.41	4.82	2.94	2.35

Table 5: Chaff results for Braun multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard "time"-utility.

A Usage instructions

This package contains a generator of factorial benchmarks for different solvers. It is able to produce problem instances in smodels- and dimacs- formats. This package has been tested to work under Debian Linux operating system, but it is expected to work also under different flavors of Unix. It uses GNU multiprecision library, which has to be installed prior to installing this package. To use this package, you need at least version 3.0 of this library. To build and install package use commands

```
> make
> make install
```

Once installed, you are able to create problem instances using the command

```
> genfacbm format type design width seed
```

The parameters are

- The format of the generated instance. Supported formats are bcsat, smodels and dimacs.
- The type of instance. Supported types are unsat(isfiable), sat(isfiable) and 4,8,16,32,64,128,256 and 512. The numbers correspond to generating an instance with the specified amount of solutions.

width	0	2	4	8	16	32	64	128	256	512
16	0.78	0.28	0.84	0.29	0.25	0.23	0.00	0.00	0.00	0.00
17	1.32	0.96	1.50	0.33	0.30	0.27	0.00	0.00	0.00	0.00
18	2.04	0.44	3.40	0.41	0.35	0.29	0.00	0.00	0.00	0.00
19	2.84	0.77	4.61	0.59	0.39	0.33	0.31	0.00	0.00	0.00
20	1.86	0.72	4.64	0.83	0.40	0.41	0.34	0.00	0.00	0.00
21	7.12	0.68	5.61	1.46	0.57	0.44	0.39	0.00	0.00	0.00
22	6.67	2.36	6.47	1.04	0.63	0.54	0.41	0.39	0.00	0.00
23	17.69	3.63	9.96	3.09	1.19	0.79	0.53	0.45	0.00	0.00
24	13.89	5.22	18.02	3.80	1.42	1.68	0.89	0.61	0.00	0.00
25	37.80	14.88	33.86	4.09	2.28	2.76	0.88	0.57	0.00	0.00
26	32.87	14.16	36.96	6.91	2.38	3.58	1.18	0.72	0.58	0.00
27	96.15	45.55	9.33	17.09	5.35	4.29	2.58	0.85	0.86	0.00
28	78.30	15.00	35.66	14.58	6.31	5.06	2.05	1.00	0.77	0.00
29	225.52	64.33	26.86	32.95	13.06	9.73	3.97	1.93	1.01	0.00
30	242.79	55.04	69.24	42.69	15.70	12.12	8.98	2.72	1.34	0.00
31	788.57	103.74	57.92	80.05	33.48	24.47	11.14	4.29	2.40	0.90
32	1034.32	225.70	0.00	114.50	31.88	29.51	15.79	5.25	1.71	1.10

Table 6: Chaff results for adder tree multiplier. Width on rows is the width of the product. The labels on columns are the number of the solutions. The values in table are seconds measured with Linux standard “time”-utility.

- Multiplier circuit design. Supported designs are braun and atree.
- The width of the instance gives the size of the number to be factored in bits. This is main factor in the expected computational effort required to solve the generated instance.
- There exists large amount of numbers corresponding different instances of specified class. Seed selects one of them. An instance can be reproduced by using the same seed. To generate different instances use different seeds.

Finally the instance will be printed to the default output stream.

It is recommended to use this command in empty directories, as it will silently overwrite existing files with names used by the program.

If you experience problems with this package, you may contact the author Tuomo Pyhälä (Tuomo.Pyhala@hut.fi).

The bc2cnf utility distributed in this package is made by Tommi Junttila (Tommi.Junttila@hut.fi).